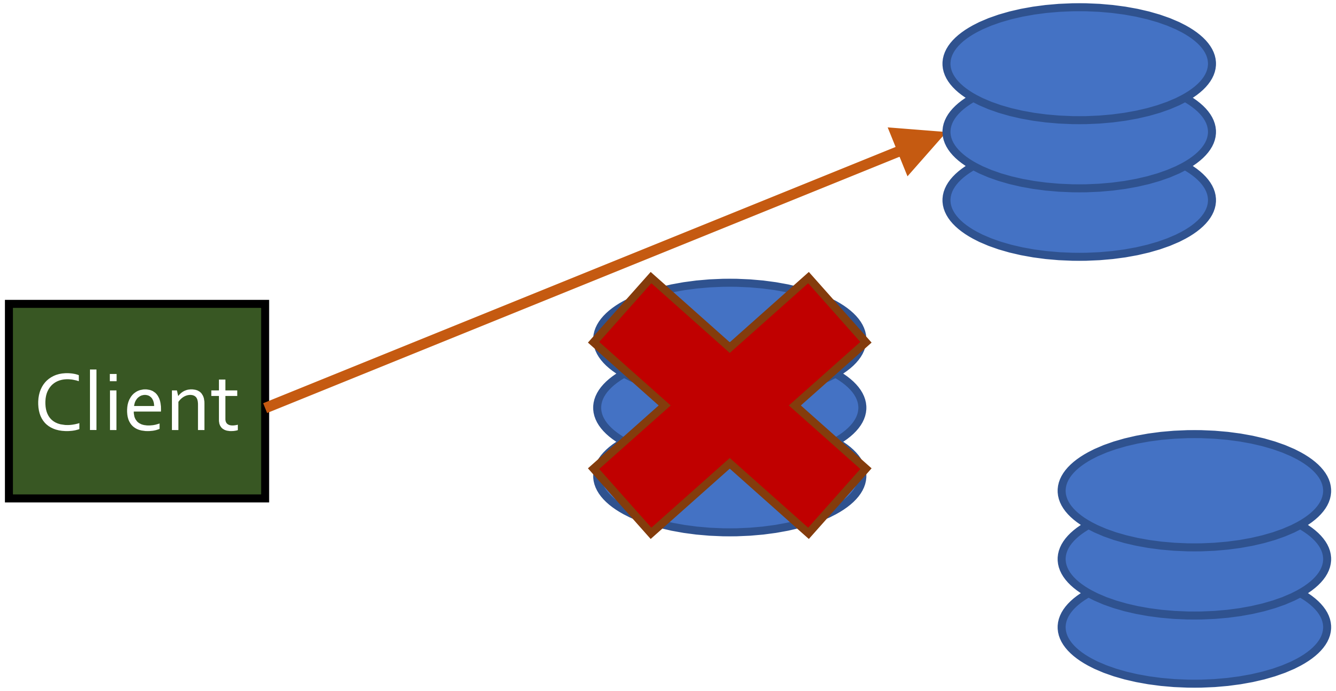
Gryff: Unifying Consensus and Shared Registers

Matthew Burke
Cornell University

Audrey Cheng
Wyatt Lloyd
Princeton University

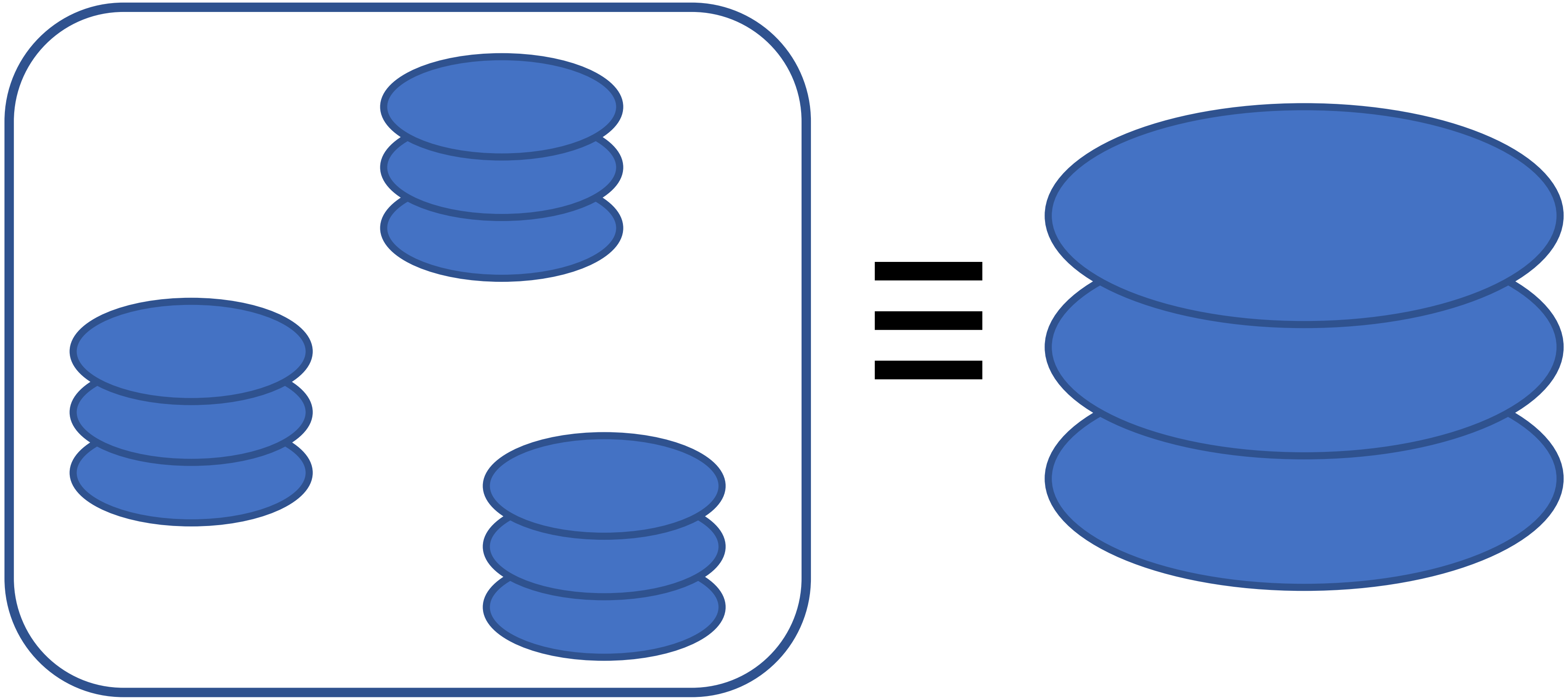
Applications Rely on Geo-Replicated Storage

- **Fault tolerant**: data is safe despite failures



Applications Rely on Geo-Replicated Storage

- **Fault tolerant**: data is safe despite failures
- **Linearizable**: intuitive for application developers



Linearizable Replicated Storage Systems

etcd

Cockroach DB

Cloud Spanner

Azure

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Status Quo: Consensus *or* Shared Registers

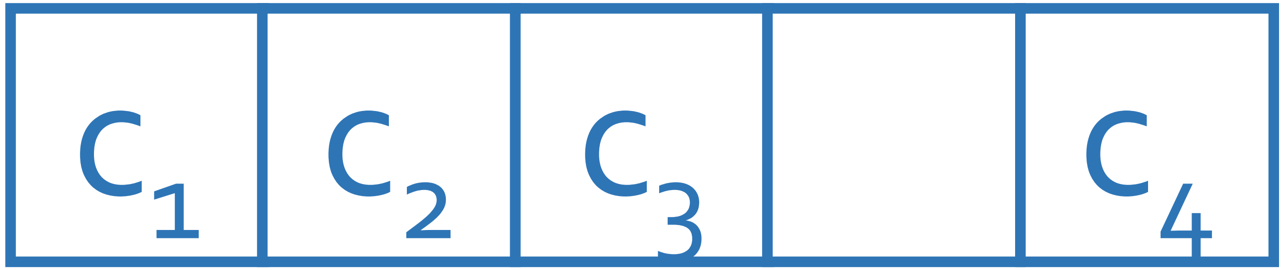
- Given the desire for **fault tolerance** and **linearizability**

	Consensus	Shared Registers
Strong Synchronization	✓	✗
Low Read Tail Latency	✗	✓

Unify consensus and shared registers?

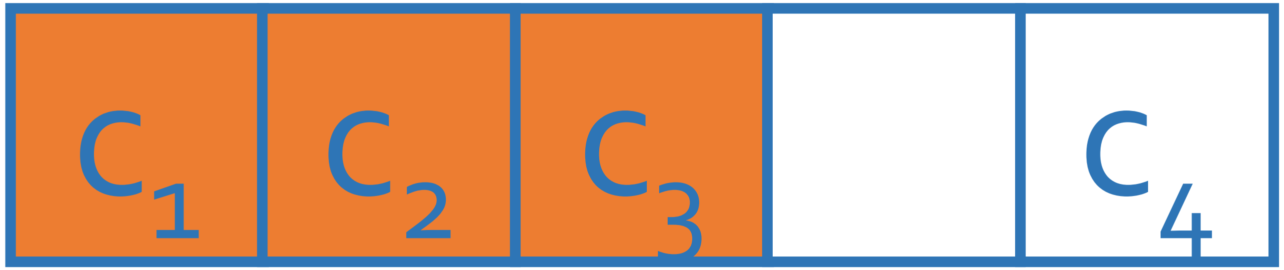
Consensus & State Machine Replication (SMR)

- Generic interface: `Command(c(.))`
- **Stable ordering**: all preceding log positions are assigned commands



Consensus & State Machine Replication (SMR)

- Generic interface: `Command(c(.))`
- **Stable ordering**: all preceding log positions are assigned commands
- Used in etcd, CockroachDB, Spanner, Azure Storage, Chubby



SMR Requires Stable Order

- Allow for strong synchronization primitives like read-modify-writes
- High tail latency in practice (e.g., by serializing through a leader)

	Consensus
Strong Synchronization	✓
Low Read Tail Latency	✗

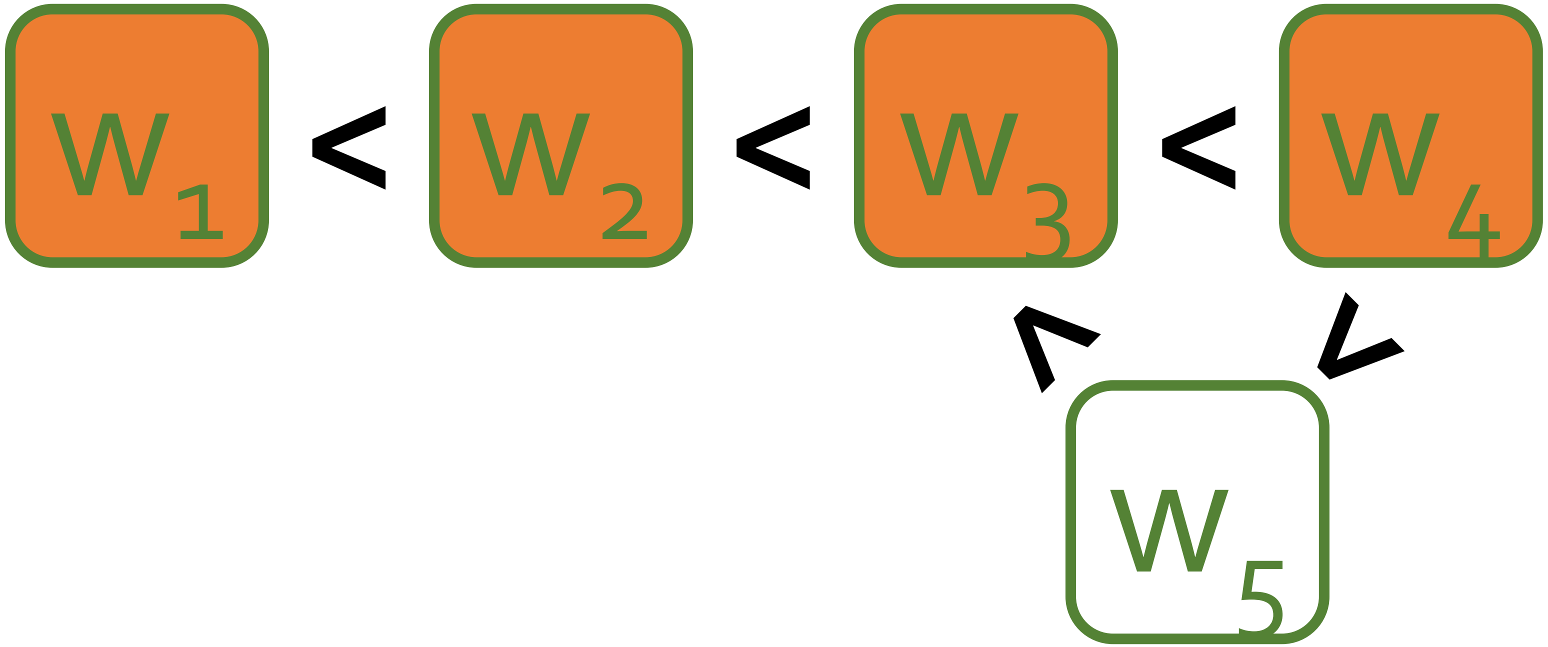
Shared Registers

- Simple interface: `Read()`/`Write(v)`
- **Unstable ordering**: total order without pre-defined positions

$$W_1 < W_2 < W_3 < W_4$$

Shared Registers

- Simple interface: `Read()`/`Write(v)`
- **Unstable ordering**: total order without pre-defined positions



Shared Registers

- Simple interface: `Read()`/`Write(v)`
- **Unstable ordering**: total order without pre-defined positions
- Similar to Cassandra, Dynamo, Riak

Shared Registers Use Unstable Order

- Cannot implement strong synchronization primitives [Herlihy91]
- Flexibility of unstable order provides favorable tail latency

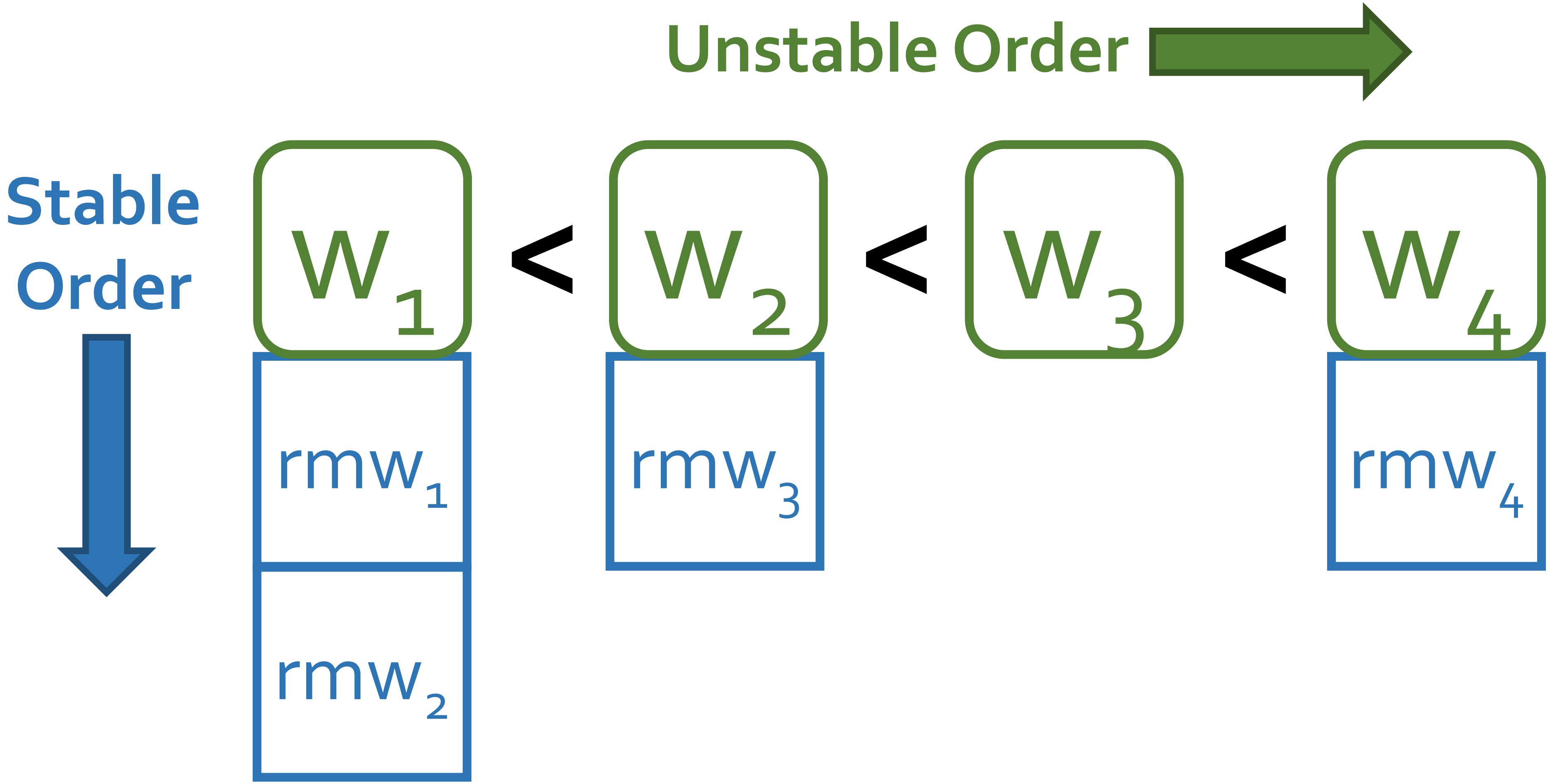
	Consensus	Shared Registers
Strong Synchronization	✓	✗
Low Read Tail Latency	✗	✓

Shared Objects: Interface for Unification

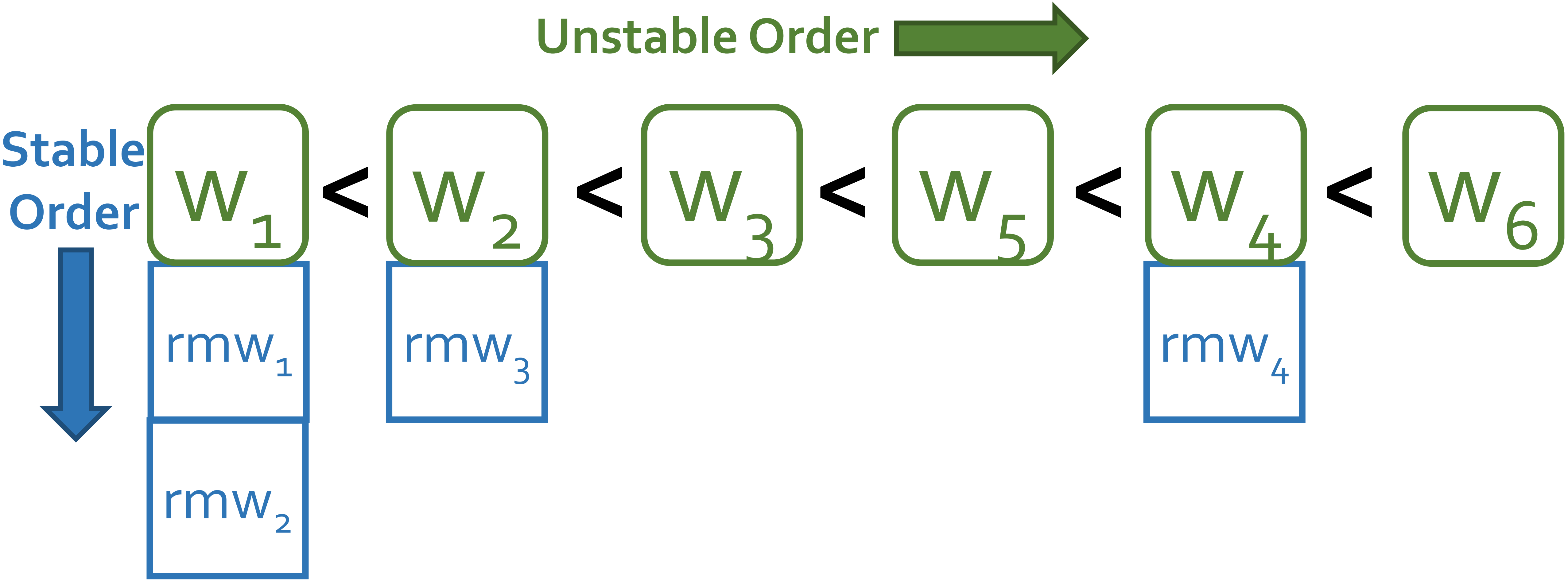
- Interface: `Read()`/`Write(v)`/`RMW(f(.))`
- `RMW(f(.))` → read base *v*, compute new value *f(v)*, write *f(v)*
- Examples: etcd, Redis, BigTable

RMWs with low read tail latency?

Consensus-after-Register Timestamps (Carstamps)

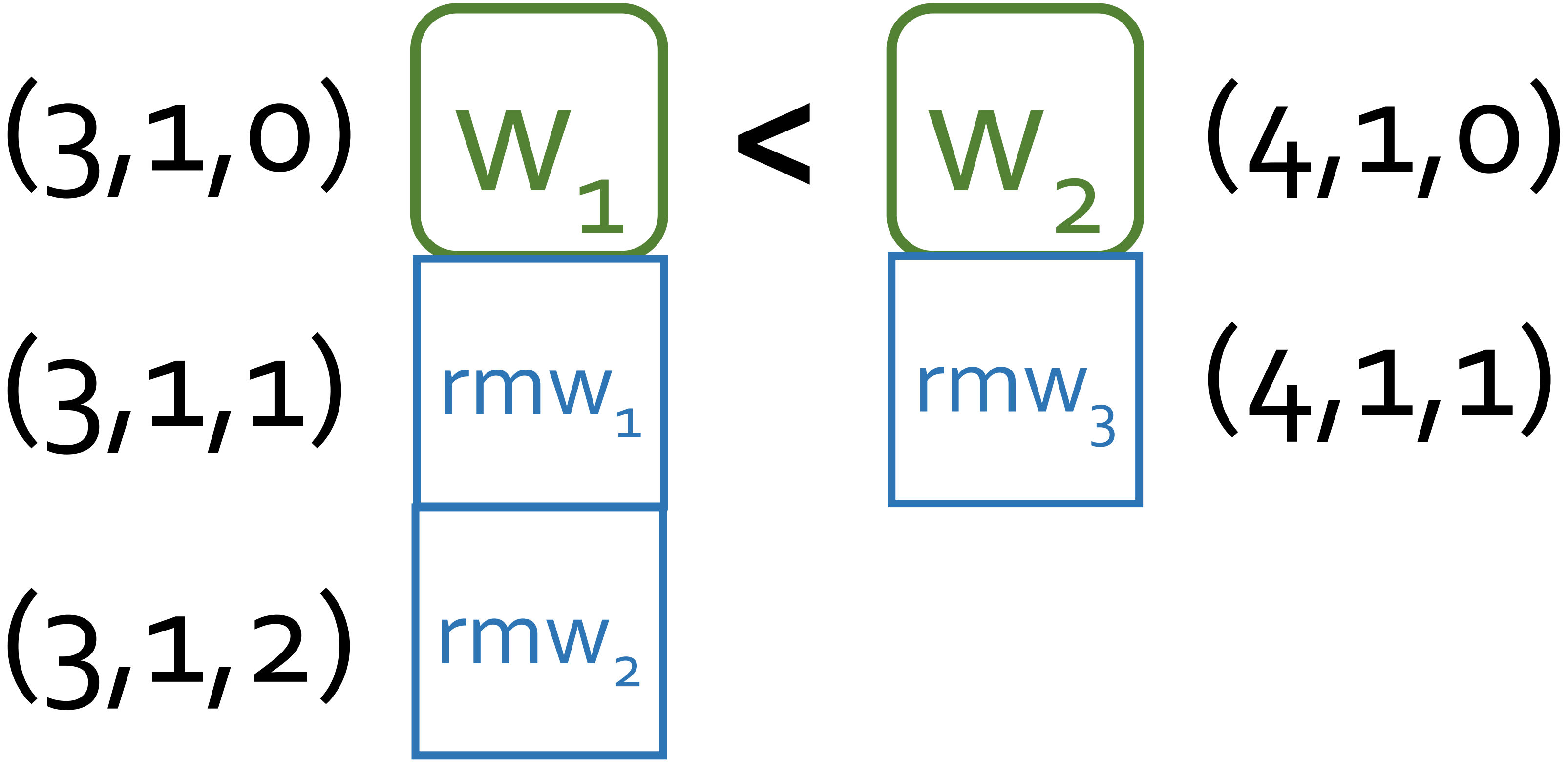


Consensus-after-Register Timestamps (Carstamps)



Carstamps

- Tuple with three fields: (*ts*, *id*, *rmwc*)
- *ts* and *id* basis for unstable ordering of writes
- *rmwc* is set to 1 greater than rmwc of base to ensure stable ordering



Gryff Unifies Consensus and Shared Registers

- Only uses consensus when necessary, for strong synchronization

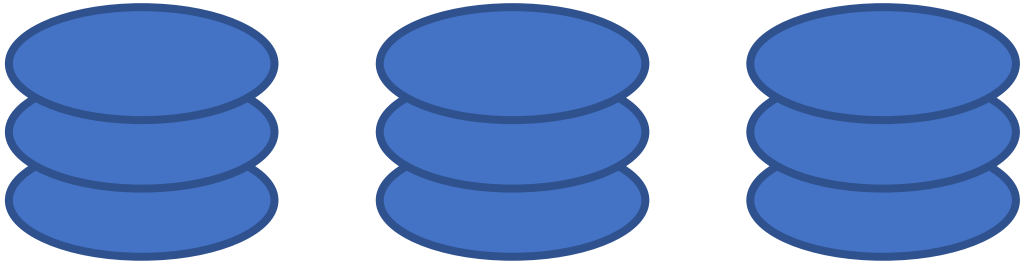
	Consensus	Shared Registers	Gryff
Strong Synchronization	✓	✗	✓
Low Read Tail Latency	✗	✓	✓

Gryff Design

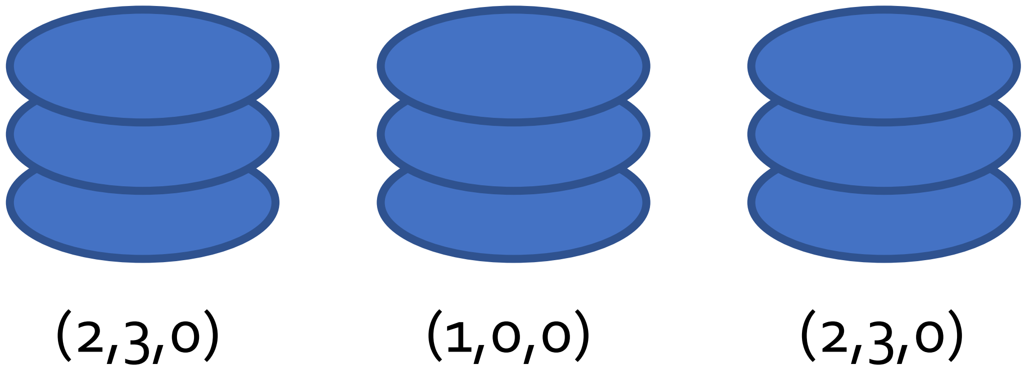
- Combine multi-writer [LS97] ABD [ABD95] & EPaxos [MAK13]
- Modifications needed for safety:
  - Carstamps for proper ordering
  - Synchronous Commit phase for rmws
- Modifications for better read tail latency:
  - Early termination for reads (fast path)
  - Proxy optimization for reads (fast path more often)

See the paper for details!

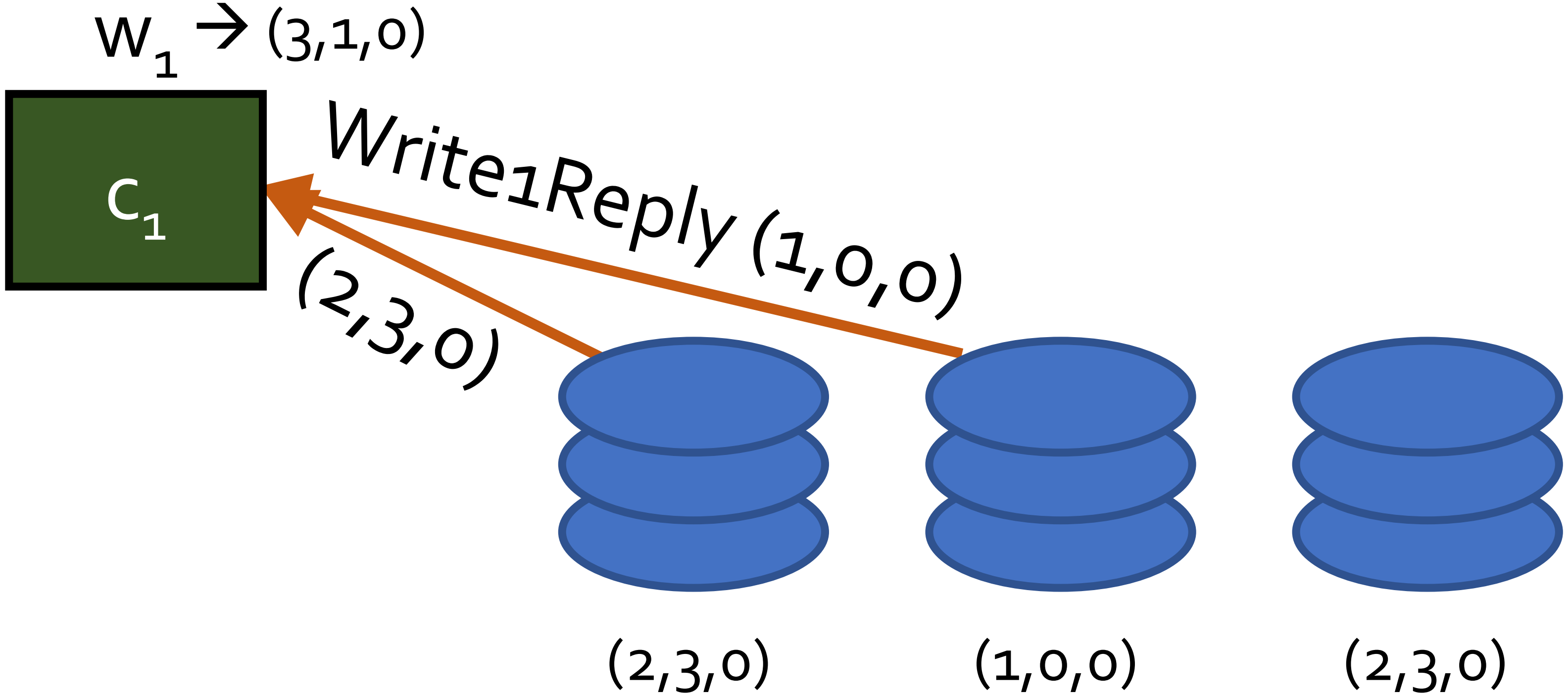
Gryff in Action



Gryff in Action

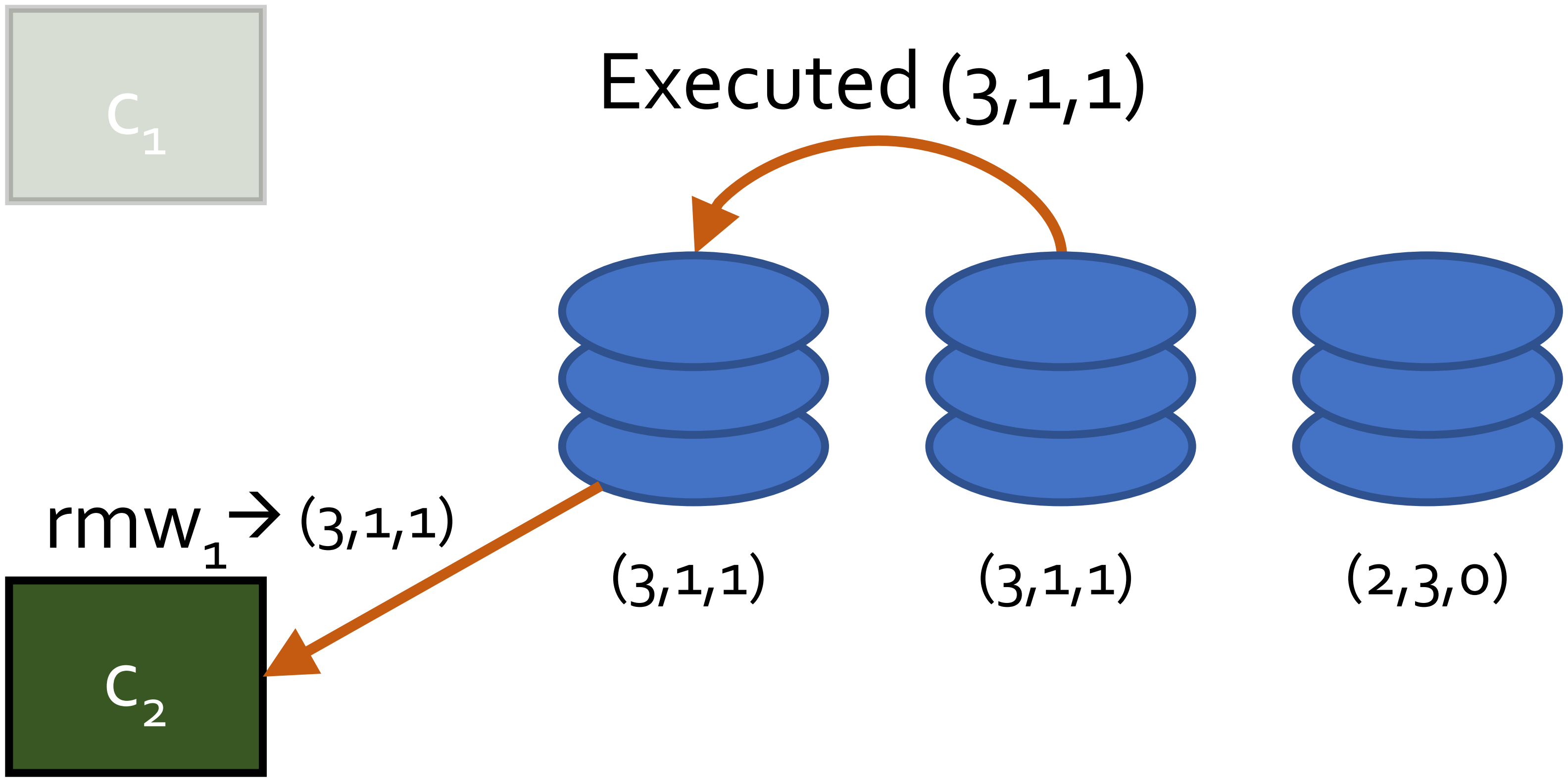


Gryff in Action



Writes always terminate in 2 phases

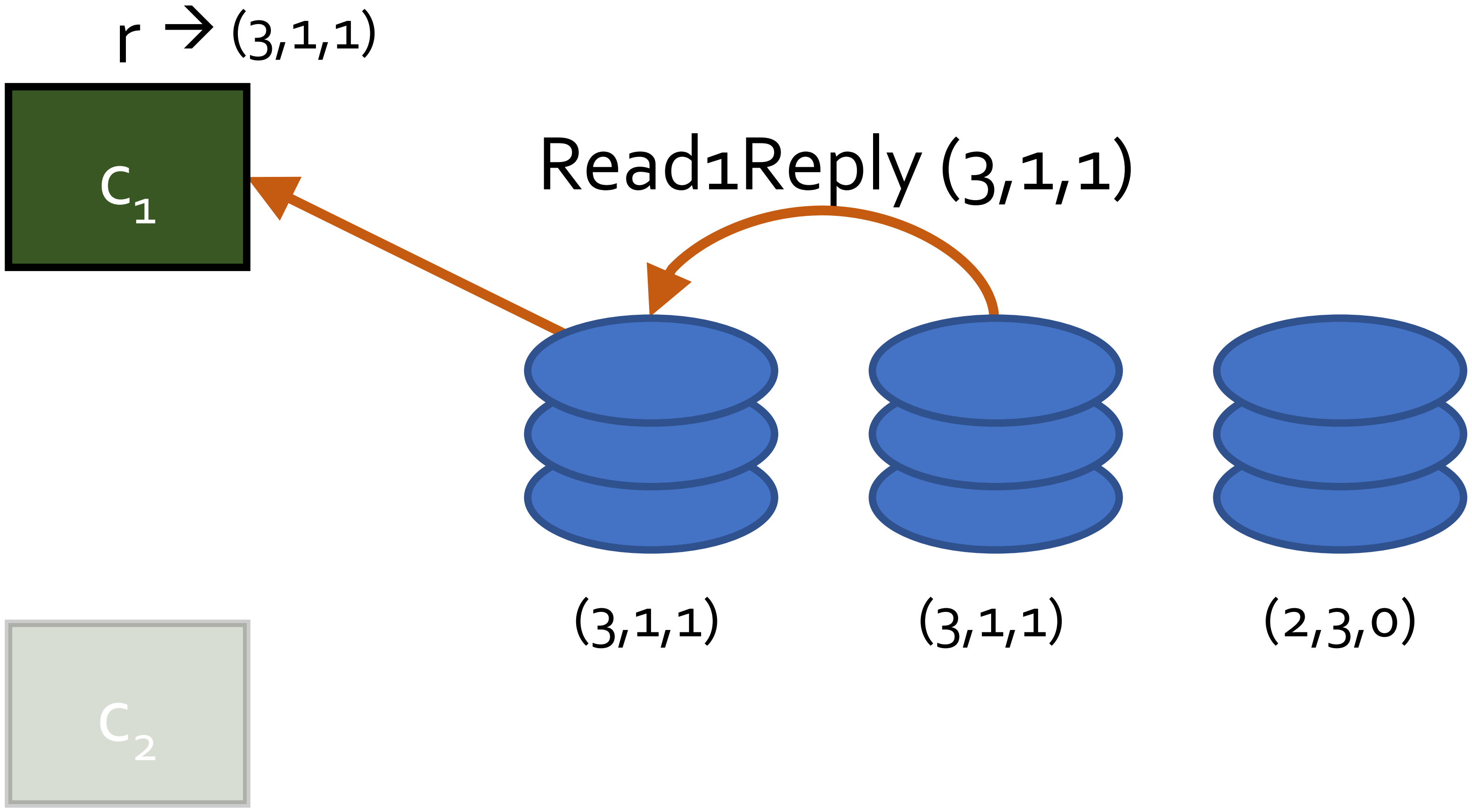
Gryff in Action



Writes always terminate in 2 phases

RMW carstamps directly after base

Gryff in Action



Writes always terminate in 2 phases

RMW carstamps directly after base

Reads often terminate in 1 phase

Evaluation

Relative to state-of-the-art-consensus protocols:

1. How do Gryff's read/write protocols affect **read tail latency**?
2. What is the **latency distribution** of Gryff's **reads, writes, and rmws**?
3. What **maximum throughput** does Gryff achieve?
4. How does Gryff perform in **tail-at-scale** workloads?

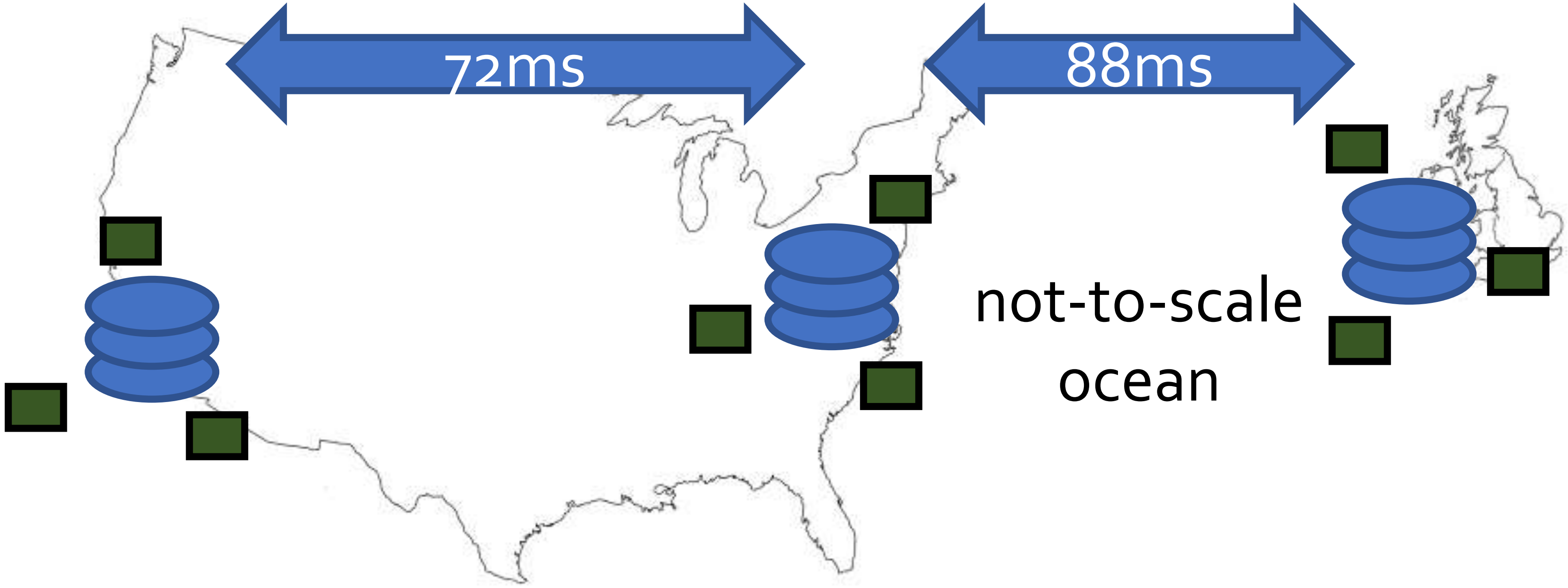
Evaluation

Relative to state-of-the-art-consensus protocols:

1. How do Gryff's read/write protocols affect **read tail latency**?
2. What is the **latency distribution** of Gryff's **reads, writes, and rmws**?
3. What **maximum throughput** does Gryff achieve?
4. How does Gryff perform in **tail-at-scale** workloads?

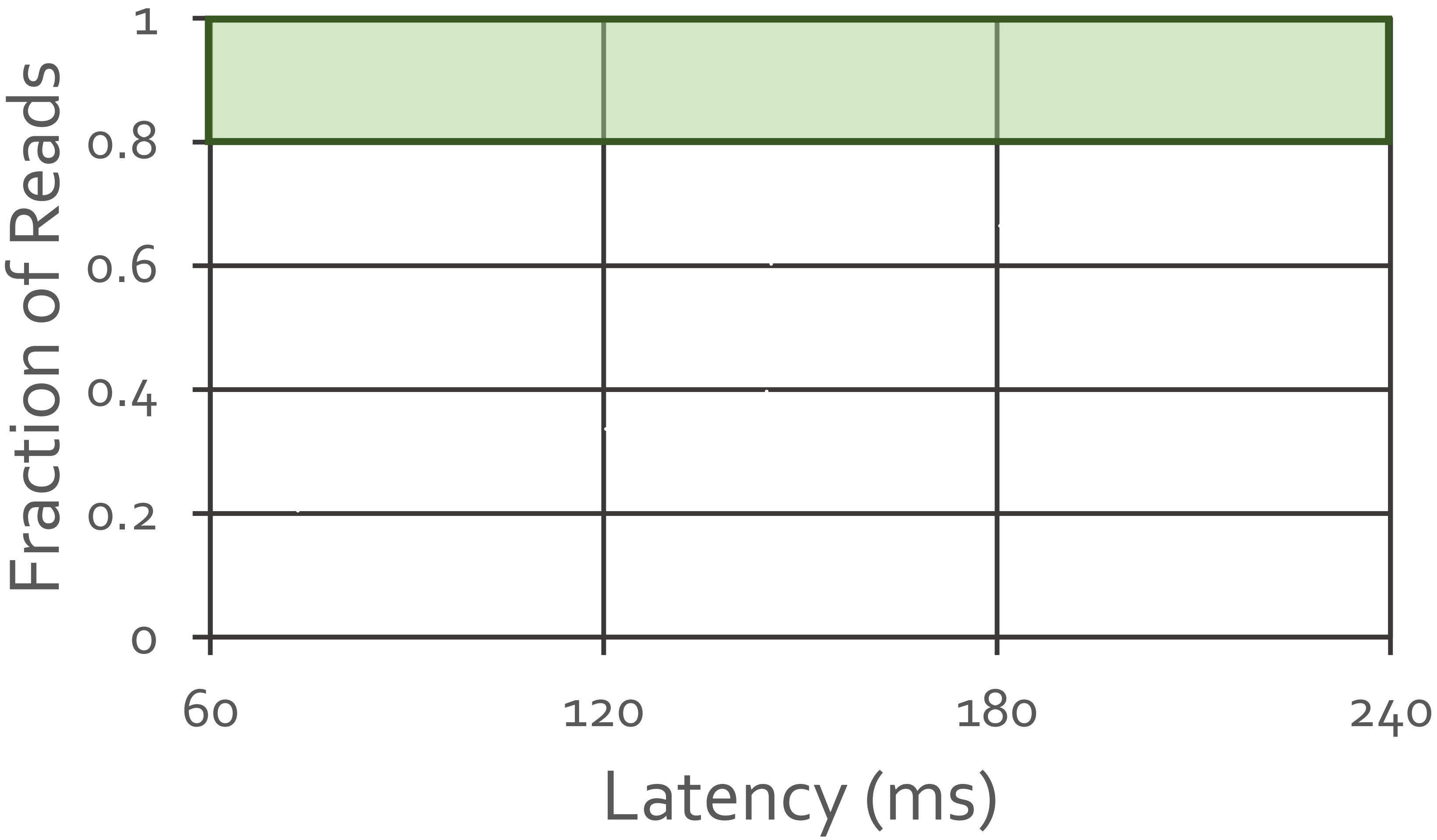
Evaluation Setup

- Geo-replication with 3 regions

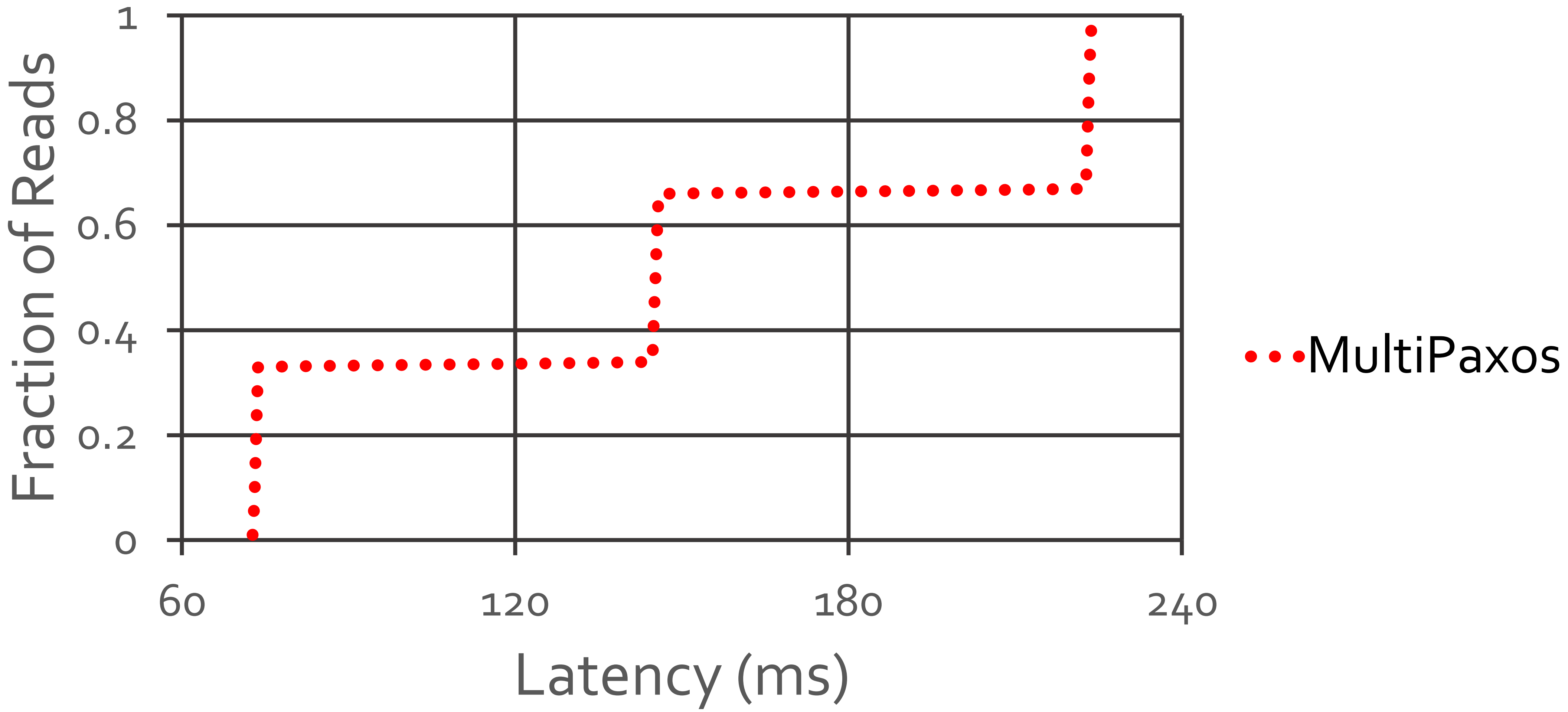


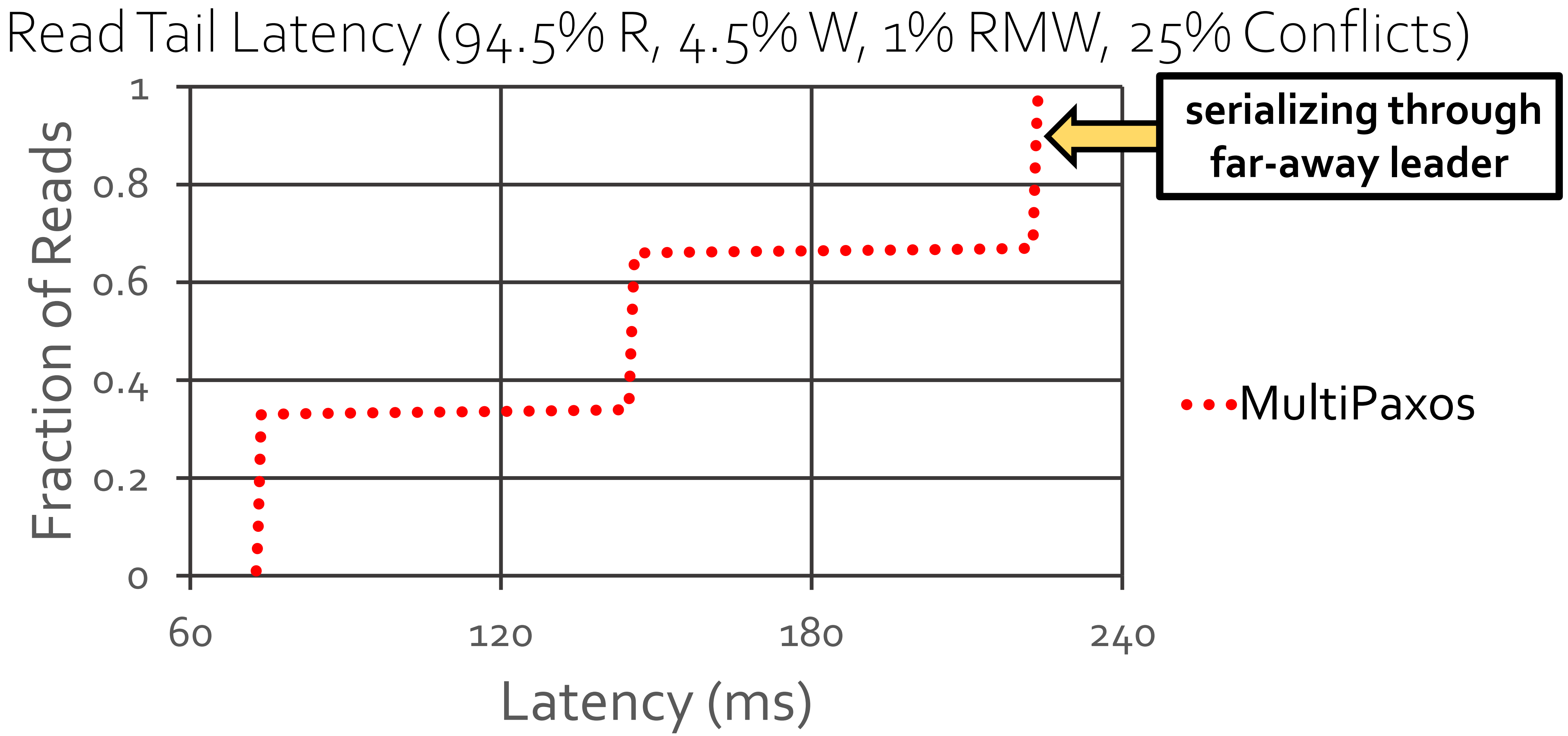
- Baselines: MultiPaxos (industry standard), EPaxos (leaderless)

Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



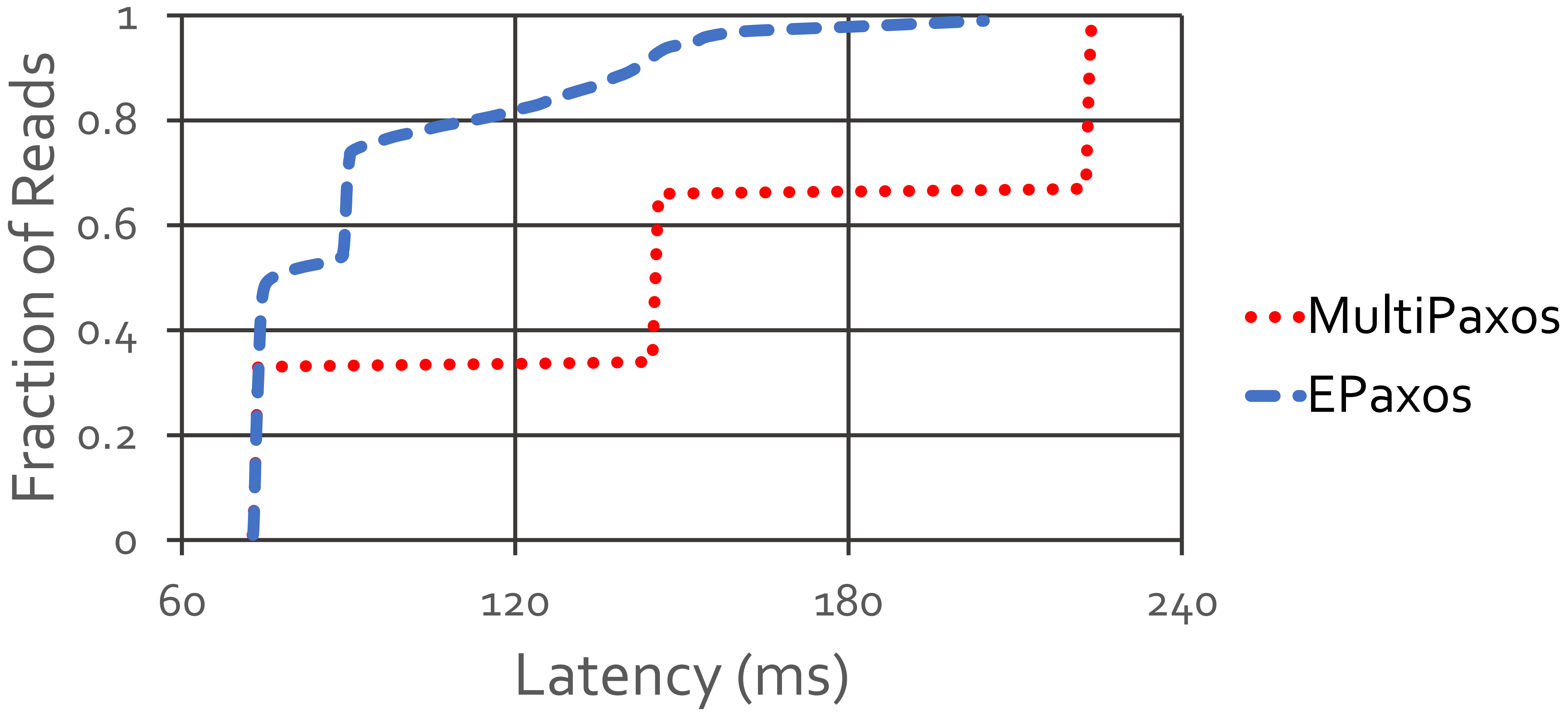
Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



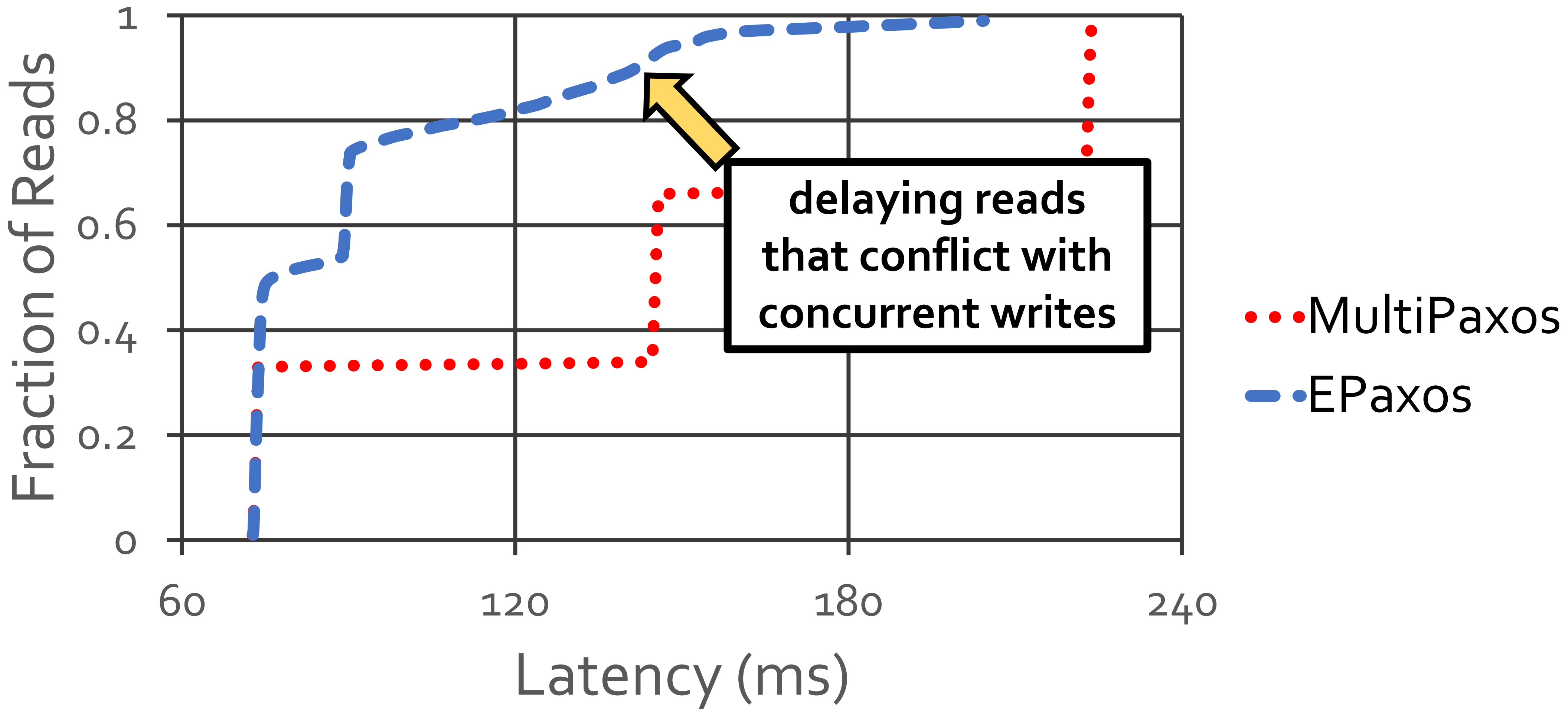


Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)
Fraction of Reads
1
0.8
0.6
0.4
0.2
0
60
120
180
240
Latency (ms)
serializing through far-away leader
MultiPaxos

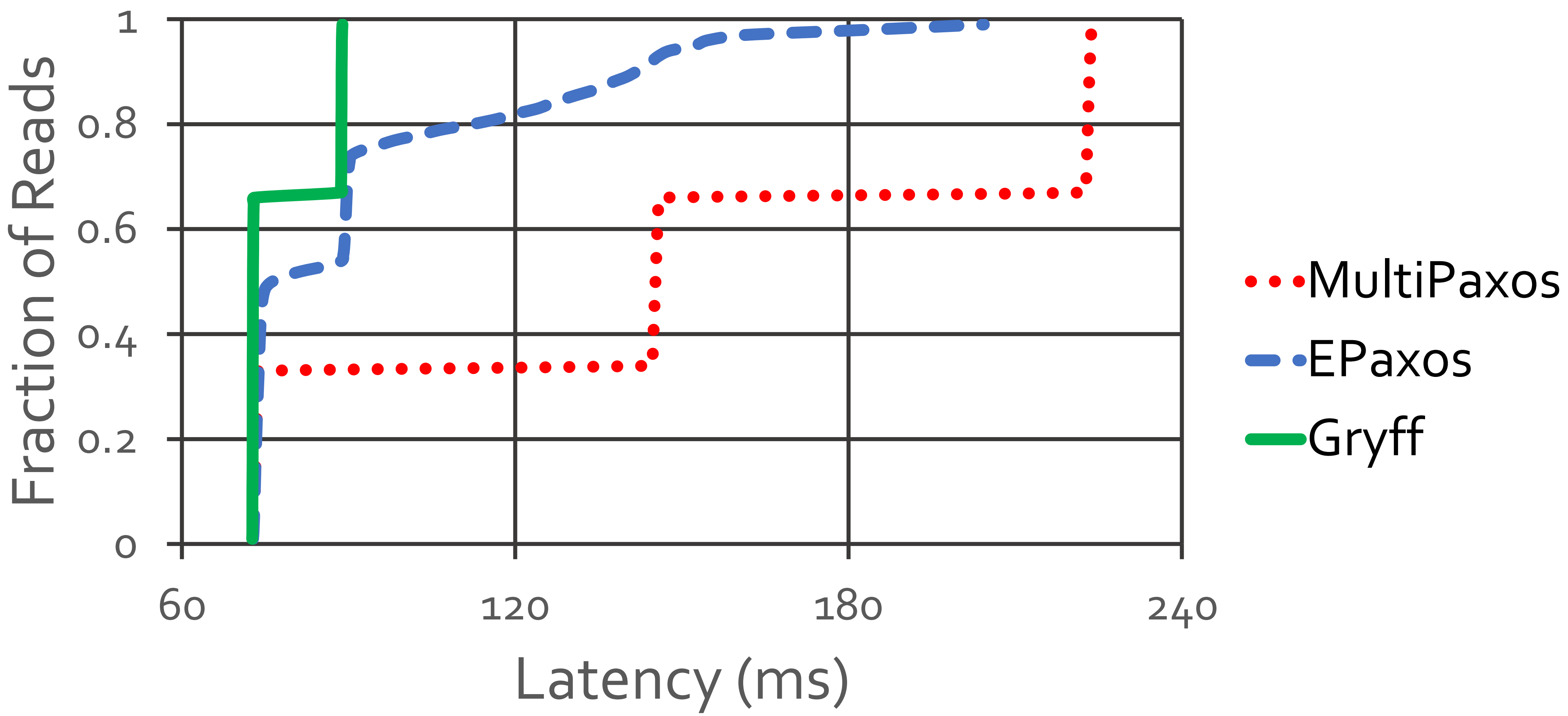
Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



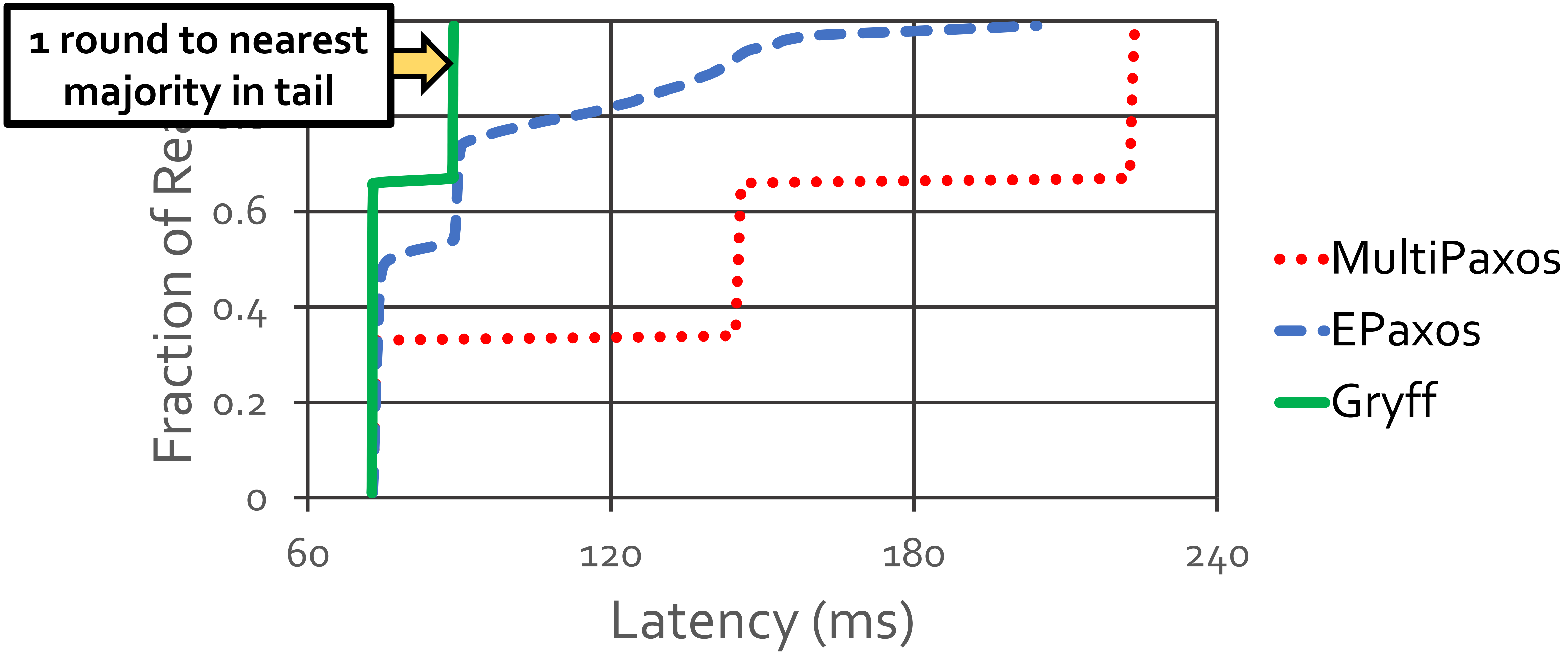
Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



Read Tail Latency (94.5% R, 4.5% W, 1% RMW, 25% Conflicts)



Summary

- Consensus: strong synchronization w/ high tail latency
  Shared registers: low tail latency w/o strong synchronization

- **Carstamps** stably order read-modify-writes within a more efficient unstable order for reads and writes

- **Gryff** unifies an optimized shared register protocol with a state-of-the-art consensus protocol using carstamps

- Gryff provides strong synchronization w/ low read tail latency

Image Attribution

- Griffin by Delapouite / CC BY 3.0 Unported (modified)
- etcd
- CockroachDB
- Spanner by Google / CC BY 4.0